

Code Complete 2: Realities of Modern Software Construction

www.construx.com

© 2004 Construx Software Builders, Inc.
All Rights Reserved.

Construx[®]

Enabling Software Project Success™

CODE Really, Really COMPLETE

A Practical
Handbook of
Software
Construction



STEVE McCONNELL



DA VINCI CODE COMPLETE



Steve McConnell

CODE COMPLETE

A Practical
Handbook of
Software
Construction

***THIS TIME IT'S
PERSONAL***



STEVE McCONNELL



Microsoft

CODE COMPLETE

2
Second Edition



A practical handbook of software construction

Steve McConnell

Available June 7



Code Complete Mission

- ❖ Attempt in 1993 was to capture lasting knowledge of software construction
- ❖ I've asserted for many years that 95% of the content of CC1 is still relevant
- ❖ Was this true?



Scope of Work for CC2

- ❖ Formally inspected entire first edition
- ❖ ~500 programming examples updated to Java, VB, C++
- ❖ New chapters on Design, Classes, Defensive Programming, Collaborative Construction, Refactoring
- ❖ OO & web integrated throughout
- ❖ Further Reading updated throughout
- ❖ Numerous complementary resources on companion website cc2e.com



Scope of Work for CC2

- ❖ Formally inspected entire first edition
- ❖ ~500 programming examples updated to Java, VB, C++
- ❖ New chapters on Design, Classes, Defensive Programming, Collaborative Construction, Refactoring
- ❖ OO & web integrated throughout
- ❖ Further Reading updated throughout
- ❖ Numerous complementary resources on companion website cc2e.com



Scope of Work for CC2

- ❖ Formally inspected entire first edition
- ❖ ~500 programming examples updated to Java, VB, C++
- ❖ New chapters on Design, Classes, Defensive Programming, Collaborative Construction, Refactoring
- ❖ OO & web integrated throughout
- ❖ Further Reading updated throughout
- ❖ Numerous complementary resources on companion website cc2e.com



Some of the Worst Construction Ideas of 1990s

❖ Code & fix



Some of the Worst Construction Ideas of 1990s

- ❖ Code & fix
- ❖ “All design up front” programming



Some of the Worst Construction Ideas of 1990s

- ❖ Code & fix
- ❖ “All design up front” programming



Some of the Worst Construction Ideas of 1990s

- ❖ Code & fix
- ❖ “All design up front” programming
- ❖ Design for speculative requirements



Some of the Worst Construction Ideas of 1990s

- ❖ **Code & fix**
- ❖ **“All design up front” programming**
- ❖ **Design for speculative requirements**
- ❖ **Components will solve all our construction problems**



Some of the Worst Construction Ideas of 1990s

- ❖ **Code & fix**
- ❖ **“All design up front” programming**
- ❖ **Design for speculative requirements**
- ❖ **Components will solve all our construction problems**
- ❖ **Automatic programming**



Some of the Worst Construction Ideas of 1990s

- ❖ **Code & fix**
- ❖ **“All design up front” programming**
- ❖ **Design for speculative requirements**
- ❖ **Components will solve all our construction problems**
- ❖ **Automatic programming**
- ❖ **Uninformed use of the waterfall model**



Some of the Worst Construction Ideas of 1990s

- ❖ Code & fix
- ❖ “All design up front” programming
- ❖ Design for speculative requirements
- ❖ Components will solve all our construction problems
- ❖ Automatic programming
- ❖ Uninformed use of the waterfall model
- ❖ Calling everything “object oriented”



Some of the Worst Construction Ideas of 2000s



Some of the Worst Construction Ideas of 2000s

- ❖ Code & fix
- ❖ “No design up front” programming



Some of the Worst Construction Ideas of 2000s

- ❖ Code & fix
- ❖ “No design up front” programming
- ❖ Planning to refactor later



Some of the Worst Construction Ideas of 2000s

- ❖ Code & fix
- ❖ “No design up front” programming
- ❖ Planning to refactor later
- ❖ Offshore outsourcing will solve all our construction problems



Some of the Worst Construction Ideas of 2000s

- ❖ Code & fix
- ❖ “No design up front” programming
- ❖ Planning to refactor later
- ❖ Offshore outsourcing will solve all our construction problems



Some of the Worst Construction Ideas of 2000s

- ❖ Code & fix
- ❖ “No design up front” programming
- ❖ Planning to refactor later
- ❖ Offshore outsourcing will solve all our construction problems
- ❖ Automatic programming



Some of the Worst Construction Ideas of 2000s

- ❖ Code & fix
- ❖ “No design up front” programming
- ❖ Planning to refactor later
- ❖ Offshore outsourcing will solve all our construction problems
- ❖ Automatic programming



Some of the Worst Construction Ideas of 2000s

- ❖ Code & fix
- ❖ “No design up front” programming
- ❖ Planning to refactor later
- ❖ Offshore outsourcing will solve all our construction problems
- ❖ Automatic programming
- ❖ Uninformed use of Extreme Programming



Some of the Worst Construction Ideas of 2000s

- ❖ Code & fix
- ❖ “No design up front” programming
- ❖ Planning to refactor later
- ❖ Offshore outsourcing will solve all our construction problems
- ❖ Automatic programming
- ❖ Uninformed use of Extreme Programming
- ❖ Calling everything “agile”



Worst Ideas, 1990s vs. 2000s

1990s

- ❖ Code & fix
- ❖ “All design up front” programming
- ❖ Design for speculative requirements
- ❖ Components will solve all our construction problems
- ❖ Automatic programming
- ❖ Uninformed use of the waterfall model
- ❖ Calling everything “object oriented”

2000s

- ❖ Code & fix
- ❖ “No design up front” programming
- ❖ Planning to refactor later
- ❖ Offshore outsourcing will solve all our problems
- ❖ Automatic programming
- ❖ Uninformed use of Extreme Programming
- ❖ Calling everything “agile”



A Decade of Advances in Software Construction



0. With the Theatrical Release of Lord of the Rings ...



0. With the Theatrical Release of Lord of the Rings ...

- ❖ ALL companies can have servers named Gandalf and Frodo



0. With the Theatrical Release of Lord of the Rings ...

- ❖ ALL companies can have servers named Gandalf and Frodo



1. Design has Been Raised a Level

- ❖ Programming has advanced through ability to create larger code aggregations
 - ◆ Statements
 - ◆ Routines
 - ◆ Classes
 - ◆ Packages
- ❖ Real legacy of OO might well be larger aggregations



2. Daily Build and Smoke Test

- ❖ Institutionalizes incremental integration
- ❖ Minimizes serious integration problems that used to be common
- ❖ Lots of other benefits, too



3. Standard Libraries

- ❖ **Good programmers have always used libraries**
- ❖ **Now provided with languages (Java, C++, .NET)**



4. Visual Basic

- ❖ Visual programming innovation
- ❖ The first development environment to make widespread use of COTS components
- ❖ Only language to learn Ada's syntax lessons (case statements, control statements, etc.)
- ❖ Highly integrated environment



5. Open Source Software

- ❖ Great aid to programmers during development
- ❖ Reduced barriers to making code available
- ❖ Opportunity to learn from available code
- ❖ Improved ability to *read* code
- ❖ Nice “community” of programmers



6. The Web for Research

- ❖ **FAQs**
- ❖ **Discussion groups**
- ❖ **Searchability in general**



7. Widespread Use of Incremental Development

- ❖ **Concepts were well known in 1990s**
- ❖ **Practice is well established in 2000s**

From CC1:

"The word 'incremental' has never achieved the designer status of 'structured' or 'object-oriented,' so no one has ever written a book on 'incremental software engineering.' That's too bad because the collection of techniques in such a book would be exceptionally potent."



8. Test-First Development

- ❖ Shortens time to defect detection
- ❖ Increases personal discipline
- ❖ Complements daily build & smoke test



9. Refactoring as a Discipline

- ❖ **Provides a discipline for making changes**
 - ◆ Not so good as a total design strategy
- ❖ **Good example of incrementalism**



10. Faster Computers

- ❖ Compare CC1 performance benchmarks to CC2 benchmarks
- ❖ Implications for optimization
- ❖ Implications for programming languages
- ❖ Implications for development



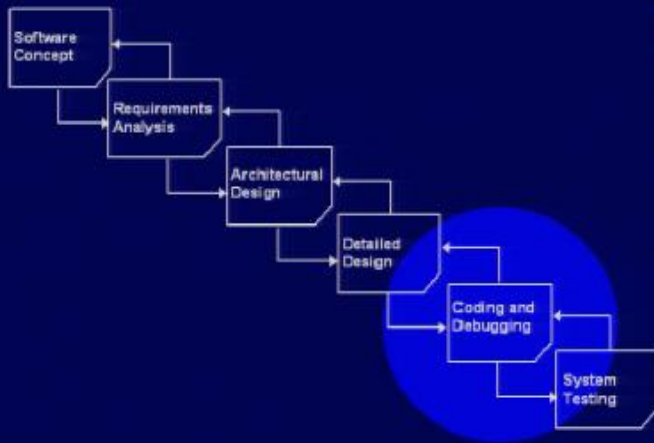
Ten Realities of Modern Software Construction

-1-

**“Construction” is a
Legitimate Topic**



Software “Construction” – Used to Look Like This





Software “Construction” – Now Looks Like This





Distinction Between Activities and Phases

- ❖ Activity != Phase (<> for VB programmers)
- ❖ Talking about “Construction” as an activity does not imply a distinct phase
- ❖ Differentiating between kinds of activities is extremely helpful

-2-

**Individual Variation Is
Significant**



Where do Variations Exist?

Researchers have found variations ranging from 10x to 28x in:

- ❖ Coding speed
- ❖ Debugging speed
- ❖ Defect-finding speed
- ❖ Percentage of defects found
- ❖ Bad-fix injection rate
- ❖ Design quality
- ❖ Amount of code generated from a design
- ❖ Etc.



Key Skills of an Expert Programmer

- ❖ Designing
- ❖ Flushing out errors and ambiguities in requirements
- ❖ Coding (naming, formatting, commenting)
- ❖ Reading & reviewing code
- ❖ Integration
- ❖ Debugging
- ❖ Unit testing
- ❖ Teamwork
- ❖ Using tools for all of the above

-3-

**Personal Discipline
Matters**



Why Personal Discipline Matters

- ❖ **Being realistic about predicting the future**
- ❖ **Areas where discipline matters**
 - ◆ Refactoring
 - ◆ Prototyping
 - ◆ Optimization
 - ◆ Minimal-complexity designs specifically
 - ◆ Managing complexity generally
- ❖ **Endpoints—Discipline and Courage**
 - ◆ Humphrey on PSP
 - ◆ Beck on Extreme Programming

-4-

**A Focus on Simplicity
Works Better than a
Focus on Complexity**



Simplicity vs. Complexity

- ❖ Why do projects fail?
- ❖ Focus on read-time convenience, not write-time convenience
- ❖ YAGNI and design for speculative requirements

-5-

**Defect-Cost Increase
is Alive and Well**



Defect Cost Increase

Activity in which a
Defect Is
Introduced

Requirements

Architecture

Construction

Average
Cost to
Correct

Requirements

Architecture

Construction

System test

Post-Release

Activity in Which a Defect Is Detected



Defect Cost Increase





Decades of Research Support Defect-Cost Increase

- ❖ Fagan, Michael E. 1976. "Design and Code Inspections to Reduce Errors in Program Development." *IBM Systems Journal* 15, no. 3: 182-211.
- ❖ Humphrey, Watts S., Terry R. Snyder, and Ronald R. Willis. 1991. "Software Process Improvement at Hughes Aircraft." *IEEE Software* 8, no. 4 (July): 11-23.
- ❖ Leffingwell, Dean. 1997. "Calculating the Return on Investment from More Effective Requirements Management," *American Programmer*, 10(4):13-16.
- ❖ Willis, Ron R., et al. 1998. "Hughes Aircraft's Widespread Deployment of a Continuously Improving Software Process," Software Engineering Institute/Carnegie Mellon University, CMU/SEI-98-TR-006, May 1998.
- ❖ Grady, Robert B. 1999. "An Economic Release Decision Model: Insights into Software Project Management." In *Proceedings of the Applications of Software Measurement Conference*, 227-239. Orange Park, FL: Software Quality Engineering.
- ❖ Shull, et al. 2002. "What We Have Learned About Fighting Defects," *Proceedings, Metrics 2002*. IEEE, pp. 249-258.
- ❖ Boehm, Barry and Richard Turner. 2004. *Balancing Agility and Discipline: A Guide for the Perplexed*, Boston, Mass.: Addison Wesley, 2004.

-6-

Design Is Important



Design Advice—What has Changed in 10 Years?



Design Advice—What has Changed in 10 Years?

- ❖ In 1990s, design pundits wanted to dot every *i* and cross every *t* before writing any code



Design Advice—What has Changed in 10 Years?

- ❖ In 1990s, design pundits wanted to dot every *i* and cross every *t* before writing any code
- ❖ In 2000s, design pundits say BDUF? YAGNI!
- ❖ There are lots of valid points on the “no design”—“all design” continuum



Design Advice—What has Changed in 10 Years?

- ❖ In 1990s, design pundits wanted to dot every *i* and cross every *t* before writing any code
- ❖ In 2000s, design pundits say BDUF? YAGNI!
- ❖ There are lots of valid points on the “no design”—“all design” continuum
- ❖ The only 2 points guaranteed to be wrong are the two being advocated!



General Point: Extremes are Usually Not Productive

- ❖ All design up front vs. no design up front
- ❖ Entirely planned vs. entirely improvised
- ❖ Pure iterative vs. straight sequential
- ❖ All structure vs. all creative
- ❖ Document everything vs. document nothing

-7-

Technology Waves Affect Construction Practices



Effect of Technology Waves on Construction

- ❖ Definition of “technology wave”
 - ◆ Early-wave characteristics
 - ◆ Mature-wave characteristics
 - ◆ Late-wave characteristics
- ❖ Construction is affected by technology—more than I thought (doh!)
- ❖ Technology can be addressed in terms of general principles

-8-

**Incremental
Approaches Work Best**



Perspective on Incrementalism

- ❖ The pure waterfall model is not at all incremental or iterative—which is why it hasn't worked very well
- ❖ Spiral development is highly incremental and iterative, which is part of why it does work well
- ❖ All projects will experience iteration at some point
- ❖ Think about *where* and *when* in your project you will get your incrementalism—cheaply, or expensively?

-9-

**The Toolbox Metaphor
Continues to be
Illuminating**



Toolbox Metaphor

- ❖ What's best? Agile? XP? Scrum? DSDM? CMM?
- ❖ Toolbox explains there's no one right tool for every job
- ❖ Different industry segments will have different tools and even different toolboxes
- ❖ What's in the Software Engineering Toolbox?
 - ◆ Best practices
 - ◆ Lifecycle models
 - ◆ Templates, checklists, patterns, examples
 - ◆ Software tools

-10-

Software's Essential Tensions Remain



Software's Essential Tensions

- ❖ **Software's essential tensions have remained unchanged for years:**



Software's Essential Tensions

- ❖ **Software's essential tensions have remained unchanged for years:**
 - ◆ Rigid plans vs. Improvisation



Software's Essential Tensions

- ❖ **Software's essential tensions have remained unchanged for years:**
 - ◆ Rigid plans vs. Improvisation
 - ◆ Planning vs. Fortune Telling
 - ◆ Creativity vs. Structure



Software's Essential Tensions

- ❖ **Software's essential tensions have remained unchanged for years:**
 - ◆ Rigid plans vs. Improvisation
 - ◆ Planning vs. Fortune Telling
 - ◆ Creativity vs. Structure
 - ◆ Discipline vs. Flexibility
 - ◆ Quantitative vs. Qualitative



Software's Essential Tensions

- ❖ **Software's essential tensions have remained unchanged for years:**
 - ◆ Rigid plans vs. Improvisation
 - ◆ Planning vs. Fortune Telling
 - ◆ Creativity vs. Structure
 - ◆ Discipline vs. Flexibility
 - ◆ Quantitative vs. Qualitative
 - ◆ Process vs. Product
 - ◆ Optimizing vs. Satisficing



Software's Essential Tensions

- ❖ **Software's essential tensions have remained unchanged for years:**
 - ◆ Rigid plans vs. Improvisation
 - ◆ Planning vs. Fortune Telling
 - ◆ Creativity vs. Structure
 - ◆ Discipline vs. Flexibility
 - ◆ Quantitative vs. Qualitative
 - ◆ Process vs. Product
 - ◆ Optimizing vs. Satisficing
- ❖ **Balance wavers, but basic tensions are constants**



Construx

Delivering Software Project Success

- ❖ **Training**
- ❖ **Software Projects**
- ❖ **Coaching & Consulting**

❖ ***info@construx.com***

❖ ***www.construx.com***